The Patent Office

INVESTOR IN PEOPLE

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

I the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1985 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed M Parkin

Dated 5 June 2000

27APR99 E442924-1 D02846
P017700 0.00 - 9909615.8

Patents Form 1/77

The Patent Office
Cardiff Road
Newport
NP9 1RH

Request for grant of a patent

1.	Your Reference	IMR/CEE/P975
2.	Application number	9909615.8
		27 APR 1999
3.	Full name, address and postcode of the or each Applicant Country/state of incorporation (if applicable)	Victoria University of Manchester Oxford Road Manchester M13 9PL 80665700 1 Incorporated in: United Kingdom
4.	Title of the invention	EXCEPTION HANDLING METHOD FOR USE IN PROGRAM CODE CONVERSION
5.	Name of agent	APPLEYARD LEES
	Address for service in the UK to which all correspondence should be sent	15 CLARE ROAD HALIFAX HX1 2HY
	Patents ADP number	AA005

6. Priority claimed to:

Country	Application number	Date of filing

7. Divisional status claimed from:

Number of parent application	Date of filing
-	-

8. Is a statement of inventorship and of right to grant a patent required in support of this application? YES

EXCEPTION HANDLING METHOD FOR USE IN PROGRAM CODE CONVERSION

The present invention relates in general to the field of program code conversion for computer systems and in particular, but not exclusively, to the field of emulation using dynamic binary translation.

A program for a computer takes several different forms. Typically a program is written first in a high-level language readily understood by a human programmer. The program is compiled from the high-level language into a low-level language more appropriate for control of a computer's processor and related components. However, in order for the processor to function the program code must be provided in a machine-readable form that directs primitive operations such as loading, shifting, adding and storing operations. To run faster some microprocessors use an expanded set of instructions each of which represent a sequence of primitive instructions. This is known as a "complicated instruction set computer" (CISC). However, program code written (or compiled) specifically for a first processor with a particular instruction set in most cases cannot run on any other type of processor because of differences between the instruction set for each type of processor.

An emulation program allows program code written for a processor of a first type (a "subject" processor) to be run on a processor of a second type (a "target" or "host" processor). One form of this emulation process is known as binary translation because executable binary code

appropriate to the subject processor is translated into executable binary code appropriate to the target processor.

In static binary translation an entire program is translated prior to execution of the translated program on the target processor. This involves a significant delay. Therefore, emulators have been developed which employ dynamic binary translation to translate small sections of a source program for execution immediately on the target processor. This is much more efficient because large sections of the source code will not be used in practice, or will be used only rarely. An emulator employing a dynamic translation system selects only the required parts of the source program for translation on demand, as the program is run.

There now follows a brief summary of dynamic binary translation for an emulator as may be employed in a preferred embodiment of the present invention. More detailed background in the field of dynamic binary translation is given in the applicant's co-pending application number GB 98 22075.9 entitled "PROGRAM CODE CONVERSION" the content of which is incorporated herein by reference to avoid wasteful duplication.

When performing dynamic binary translation the emulator appears to the subject code as if the subject code were running on the appropriate subject processor. The emulator replicates the subject processor including, for example, registers of the processor. The emulated registers are termed herein "abstract registers" and correspond to the set of registers of the subject

processor used by the subject code. In the preferred dynamic translation process the emulator first translates a predetermined small section of the subject code into an intermediate representation which represents the instructions of the subject code in a generic format, optionally performs optimisation on the intermediate representation, and then translates the optimised intermediate representation into executable binary code for the target processor. It is preferred that the small section of subject code corresponds to a "basic block" which starts with a first instruction at a unique entry point and ends at a last instruction at an unique exit point. Typically the last instruction of the block is a jump, call or branch instruction (conditional or unconditional).

In the field of binary translation a problem arises in he handling of exceptions. An exception indicates that a condition has occurred which needs to be handled before processing can continue. This includes explicit exceptions performed as an instruction in the subject code (for example an exception is reported if the value of one register is greater than the value of a second register), and implicit exceptions which occur for example as a result of memory read or write operations to a memory page that is not currently available. In both cases the exception is desirably reported to an exception handler written in subject code. However, many subject machine architecture definitions require that the exception is reported on a boundary between subject code instructions, following a predetermined set of rules. For example, the subject architecture definition may require that when the exception is reported, the effects of all previous subject

instructions are complete, the exception points to the first instruction of the subject code which has not been executed and no effects from that subject instruction or any subsequent instruction have yet taken place. Further, the architecture definition of a particular processor may have different rules for different types of exceptions.

In the context of binary translation it is apparent that a target instruction performed on the target processor that causes an exception to be reported will not of itself fulfil the conditions for reporting the exception to an exception handler written in subject code. Instructions are almost always performed on the target processor in a different order to the order of instructions in the corresponding block of subject code, firstly due to the differences between the instruction set of the subject processor for which the subject code was written and the target processor on which the translated target code is run, and secondly because of the optimisation of the intermediate representation that typically occurs during translation.

Exceptions can occur in response to execution of the translated target code on the target processor, and can occur during execution of the emulator code on the target processor, i.e. during translation. In order to report an exception to the subject exception handler the state of the virtual subject processor represented by the emulator must be available to the subject exception handler, including the correct status of the registers of the subject processor.

One approach to this problem is to return the virtual subject machine to the conditions that applied at entry into the section of code being translated or executed, i.e. by returning the virtual subject machine to the condition prevailing at the point of entry into the current block of subject code instructions being translated or executed. The exception handler can now step through the instructions of the block of source code individually in sequence until the instruction causing the exception is identified.

US 5832205 (Kelly et al) discloses an emulator which uses a set of "working" registers during emulation of each section of subject code. The content of each of these working registers is copied to a set of "official" virtual subject registers at the end of the section of subject code, using a gated store buffer. Therefore, if an exception occurs during emulation of a section of subject code this will affect only the working registers and the condition of the virtual subject machine can be recovered from the "official" registers at the point of entry into that section of subject code. However, the use of "working" and "official" registers adds significantly to the overhead of the emulation process in the target processor due to the copying of information from the "working" registers to the corresponding "official" registers at the end of each section of subject code.

An aim of the present invention is to provide a method of representing subject registers in an emulator which allows exceptions to be accurately reported to a subject exception handler in accordance with the rules of the handler, whilst minimising overhead in the emulator. It is

a further aim of the present invention to provide an emulation method and apparatus wherein subject registers are represented to allow accurate exception handling.

According to the present invention there is provided a method of representing a subject register in an emulator, said method comprising the steps of:

(a) mapping an abstract register representing a subject register of a subject machine to either a first location or to a second location within a target machine; and

(b) alternating mapping of the abstract register between the first and second locations such that the content of one of the first or second locations represents a definitive version of the abstract register for exception handling, whilst the other of the first or second locations represents a speculative version of the abstract register.

Preferably, for a predetermined section of source code, one of the first or second locations holds a definitive value of the abstract register at entry into that section, whilst the other of the first or second locations holds a speculative current value of the abstract register for use during that section.

Preferably, the abstract register is mapped to the alternate one of the first or second locations upon reaching the end of the section of source code. The speculative version becomes definitive when it is determined that no exception has occurred in the section

of code. Ideally, said step of alternating mapping is performed only if the content of the speculative version of the abstract register has been updated during the predetermined section of subject code.

Preferably, a plurality of abstract registers are provided each representing a register of the subject machine, each of the plurality of abstract registers is mapped to either a respective one of a first set of locations or a respective one of a second set of locations within the target machine, and the step of alternating mapping is performed for each of the abstract registers between the respective ones of the first and second sets of locations.

Preferably, the first location is a memory location. Preferably, the second location is a memory location. Optionally, the first location is a target register. Optionally, the second location is a target register.

Preferably, the method is for use with an emulator that performs dynamic binary translation, and suitably the predetermined section of source code represents one or more basic blocks of source code.

According to further aspects of the present invention there is provided an emulator method and an emulator apparatus for performing the method according to any statement herein. The present invention also extends to a computer when programmed to perform the method according to any statement herein, to a computer program for performing the method according to any statement herein, and to a computer program product containing computer

readable instructions for performing the method according to any statement herein.

For a better understanding of the invention, and to show how embodiments of the same may be carried into effect, reference will now be made, by way of example, to the accompanying diagrammatic drawings, in which:

Figure 1 shows a typical prior art configuration for a subject processor;

Figure 2 shows a typical emulator using binary translation;

Figure 3 shows a configuration of an emulator using binary translation as may be employed in preferred embodiments of the present invention;

Figure 4 shows a typical binary translation type emulator in use; and

Figures 5 and 6 show example sets of abstract registers.

Referring to Figure 1 a typical prior art arrangement is shown illustrating the configuration of a subject machine wherein subject code 10 is executed directly on a subject processor 11. Suitably the subject code is executable binary code. However, the subject code may be represented in any suitable language with intermediate layers (compilers etc.) between the subject code 10 and the processor as will be familiar to the skilled person.

Referring now to Figure 2, a typical prior art configuration is shown to illustrate the use of a binary translation type emulator 20 as an intermediate layer enabling the subject code 10 to be executed by a target processor 31. The preferred embodiment of the present invention is particularly intended for use with an emulator 20 which performs dynamic binary translation of the subject code 10 into target code 30 executable on the target processor 31.

Referring now to Figure 3 the emulator 20 of the preferred embodiment is illustrated in more detail and comprises a front end 21, a core 22 and a back end 23.

The front end 21 is configured specific to the subject processor 11 being emulated. The front end 21 translates instructions of the subject code 10 into a generic intermediate representation for each basic block of subject code. Each basic block suitably includes a sequential set of instructions between a first instructions representing a unique entry point and a last instruction at a unique exit point (such as a jump, call or branch instruction). In a particularly preferred embodiment the emulator 20 may select a group block comprising two or more basic blocks chosen for code generation and optimisation as a single unit. Further, the emulator 20 may support iso-blocks representing the same basic block of subject code under different entry conditions. Each predetermined section of the subject code 10 results in a block of intermediate representation (an "IR block").

The core 22 optimises each IR block generated by the front end 21 by employing optimisation techniques which need not be described here in detail. The back end 23 takes optimised IR blocks from the core 22 and produces target code 30 executable by the target processor 31.

As shown in Figure 4, in use a first predetermined section of the subject code 10 is identified such as a basic block 100 and translated by the emulator 20 running on the target processor 31 in a translation mode. The target processor 31 then executes the corresponding optimised and translated block 300 of target code 30.

The preferred emulator 20 includes a plurality of abstract registers, suitably provided in the core 22 shown in Figure 3, which represent the physical registers that would be used within the subject processor 11 to execute the subject code 10. The abstract registers define the state of the subject processor 11 being emulated by representing the expected effects of the subject code instructions on the subject processor registers.

As shown in Figure 5, in the preferred embodiment two sets of abstract registers are provided, here labelled set A and set B. At initialisation a first of these two sets, for example set A, holds "definitive" values. That is, the registers of set A are defined to hold initial values representing the expected content of the physical registers of the subject processor 11 being emulated which are known to be valid.

The second set of abstract registers, in this example labelled set B, are initially defined to represent

"speculative" values. That is, at initialisation the second set of abstract registers (set B) also hold the expected initial content of the physical registers of the subject processor 11, but the values in set B are not relied upon as being valid.

In the translation process the emulator 20 uses the speculative set of abstract registers (set B) such that the content is updated to show the expected state of the physical registers of the subject processor 11 after execution of the block 100 of subject code 10. During translation of the first block, the content of the abstract registers of set A remains unchanged.

If an exception occurs during translation or execution of the basic block 100 then the emulator 20 can readily recover the condition of the subject registers upon entry into the block 100 using the abstract registers marked as holding definitive data (i.e. set A). The exception handler can now step through the instructions of the block 100 of the source code 10 in sequence until the instruction causing the exception is identified. Here, the status of the abstract registers is updated after each instruction. Therefore, when the subject code instruction responsible for the exception is identified the condition of the virtual subject machine represented by the emulator can be reported to the subject code exception handler according to the rules thereof. The subject code handler will then recover the exception in accordance with the handling process and return to a point in the subject code appropriate to the exception. For example, it is common that the exception handler returns the emulator to the next unexecuted instruction of the source code 10. The

translation or execution process can then continue from that point.

After the successful execution of the translated block of target code 300 the registers in set B holding speculative values will have been updated to hold the expected content of the equivalent registers of the subject processor 11 being emulated at the end of the basic block 100 of subject code 10. At this point, the abstract registers of set A hold the condition of the subject processor at entry to the block of subject code 100 and the abstract registers of set B hold the condition at the end of the block 100. Since the block 100 has been successfully translated and executed, the abstract registers of set B are now defined as holding definitive values, and the abstract registers of set A are defined as holding speculative values.

The registers of set A and set B suitably form register pairs. Each register in set A has a corresponding partner register in set B. One of the pair holds the definitive value of that abstract register, whilst the other holds the speculative value. At the end of each section of code the definition of these two registers is reversed such that each register of the pair performs the opposite function during the next section of code. Alternating the function of the two registers of each register pair provides a simple and elegant method of maintaining the entry conditions for the current section of code.

As a further advantage, it is not necessary to update the status of every abstract register upon successful completion of each section of code. Only those registers

which have been changed during that section need be updated to show their new function. If the value of a particular abstract register is not changed during a section of code then that value will remain in place during the next section to perform the same definitive or speculative function as appropriate.

Referring now to Figures 5 and 6 a simplified example embodiment is shown. At step 1 shown in Figure 5 a first memory location (REG X_A) of an abstract register representing register X of the subject processor 11 contains the definitive value whilst the second (REG X_B) contains the speculative value. Therefore, the working map for register X points to the location of the speculative version REG X_B. Similarly, for register Y initially REG Y_A is definitive whilst REG Y_B is speculative. After performance of one or more instructions such as block 100 of the subject code 10 then the mapping for the abstract registers is updated as shown in Figure 6. In this example, the content of the speculative register Y has changed and therefore REG Y_B is now taken to be the definitive version for use in a subsequent block. The map for register Y is updated to point to REG Y_A as the speculative version. By contrast, register X was not affected by the instruction or instructions in the block 100 and therefore REG X_A remains as the definitive version whilst REG X_B remains as the speculative version.

To allow continuity of register content between sections of code, suitably the first read operation encountered during a current section of code uses the definitive version of each particular abstract register. The

definitive version represents the condition of that register at entry in to the current section of code and therefore maintains continunity with the previous section. Further read operations also use the definitive version of each abstract register, until a write operation is encountered. The first write operation uses the speculative version of each particular abstract register. Therefore the definitive version remains unchanged and the speculative version now contains the current value of the relevant abstract register. Subsequent read and write operations use the speculative version for the remainder of that section of code.

As described above in preferred embodiments of the present invention two abstract registers are provided corresponding to each physical register of the subject processor 11. One of each pair of abstract registers contains a definitive value whilst the other contains a speculative value. The function of these two registers is readily reversed to alternate the register holding definitive content. Therefore, time consuming copying operations are avoided.

In the preferred embodiment the two sets of abstract registers are achieved by mapping to two sets of memory locations and the definitive and speculative versions alternated by alternating the memory mapping between these two locations. Updating the map of the abstract registers held in the target machine to replicate the physical registers of the subject processor is performed simply at translation time, and imposes no overhead when the translated code is executed, possibly many times.

In a further preferred embodiment, one or both of the definitive and speculative versions of the abstract registers may be stored in a pair of target machine registers (on a target machine with a sufficiency thereof) as an alternative to using a pair of memory locations.

Although the embodiments described above refer to an emulator employing dynamic binary translation, the method is also applicable to static translation where a large section of code is translated prior to execution. In static translation the section of code selected for translation typically represents a whole program or a major part of a program. However, it is still convenient to use the method described above for handling exceptions arising during translation and execution of the translated code, enabling exception handling to be performed at least from the condition at entry into that section of code. Further, the method is applicable to program code optimisation wherein the subject machine and the target machine have the same or at least compatible instruction sets and architectures.

The present invention extends to a computer when programmed to perform the method described above, to a computer program for performing the method described above, and to a computer program product containing computer readable instructions for performing the method described above.

CLAIMS

1. A method of representing a subject register in an emulator, said method comprising the steps of:

(a) mapping an abstract register representing a subject register of a subject machine to either a first location or to a second location within a target machine; and

(b) alternating mapping of the abstract register between the first and second locations such that the content of one of the first or second locations represents a definitive version of the abstract register for exception handling, whilst the other of the first or second locations represents a speculative version of the abstract register.

2. A method as claimed in claim 1, wherein for a predetermined section of source code, one of the first or second locations holds a definitive value of the abstract register at entry into that section, whilst the other of the first or second locations holds a speculative current value of the abstract register for use during that section.

3. A method as claimed in claim 2, wherein said step of alternating mapping is performed upon reaching the end of the predetermined section of source code.

4. A method as claimed in claim 3, wherein said step of alternating mapping is performed only if the content of

the speculative version of the abstract register has been updated during the predetermined section of subject code.

5. A method as claimed in claim 1, wherein

a plurality of abstract registers are provided each representing a register of the subject machine;

each of the plurality of abstract registers is mapped to either a respective one of a first set of locations or a respective one of a second set of locations within the target machine; and

the step of alternating mapping is performed for each of the abstract between registers the respective one of each of the first and second sets of locations.

6. A method as claimed in claim 1, wherein the first location is a memory location and/or the second location is a memory location.

7. A method as claimed in claim 1, wherein the first location is a target register and/or the second location is a target register.

8. A method as claimed in claim 2, wherein said emulator performs dynamic binary translation.

9. A method as claimed in claim 8, wherein the predetermined section of source code represents one or more basic blocks of source code.

ABSTRACT

EXCEPTION HANDLING METHOD FOR USE IN PROGRAM CODE CONVERSION

A method of handling exceptions for use in an emulator performing program code conversion. Registers of a subject machine being emulated are represented by a pair of abstract registers on the target machine, suitably using memory locations of the target machine and/or any available target registers. One of the pair holds a definitive value at entry into a section of source code whilst the other holds a speculative value which is updated during translation and execution of that section of code. Exceptions are handled by recovering the conditions of the subject machine upon entry into the section of code using the definitive version of each abstract register. Conveniently, the function of the pair of registers is alternated upon successful completion of each section of source code. Advantageously, a definitive version of each register is always available for exception handling whilst avoiding time consuming copy and storing operations.

[Figure 2]

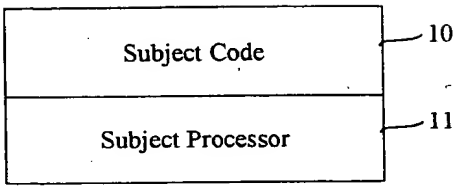

Fig. 1

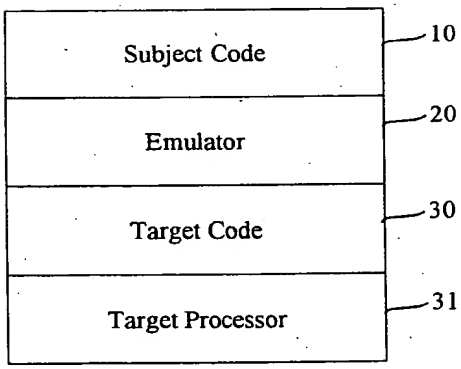

Fig. 2

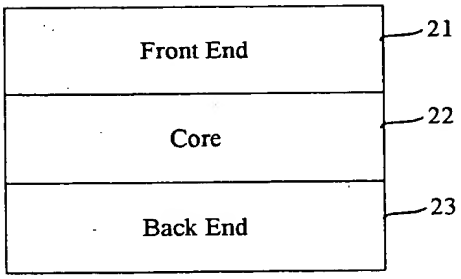

Fig. 3

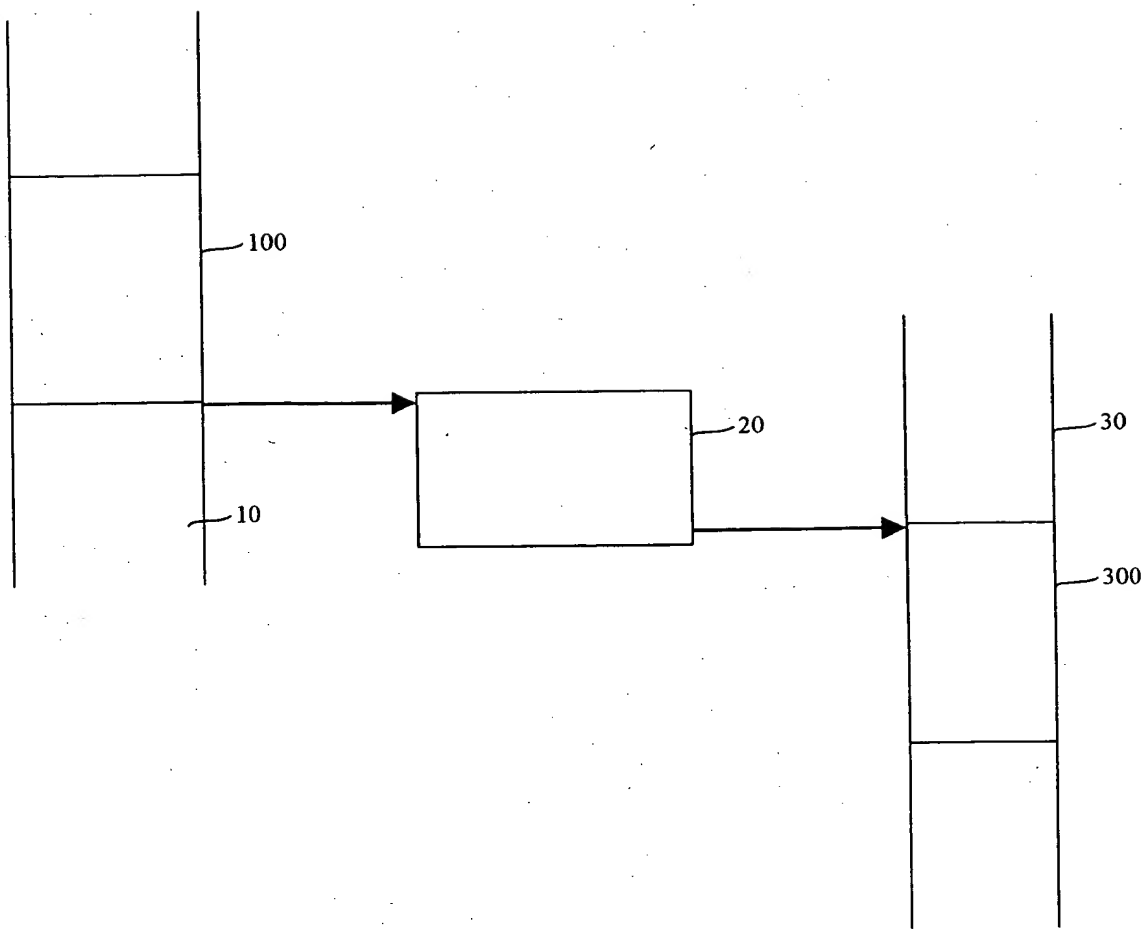

Fig. 4

Register Memory Map

Register	Target Set A Values	Target Set B Values
Reg X	Reg X_A Definitive	Reg X_B Speculative
Reg Y	Reg Y_A Definitive	Reg Y_B Speculative
...	...	...

Fig. 5

Register Memory Map

Register	Target Set A Values	Target Set B Values
Reg X	Reg X_A Definitive	Reg X_B Speculative
Reg Y	Reg Y_A Speculative	Reg Y_B Definitive
...		...

Fig. 6